Secure Programming

Tamara Rezk

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

Secure Programming

Mémoire de synthése pour l'obtention d'une

Habilitation à Diriger les Recherches

par

Tamara Rezk

soutenue le 03 Avril 2018

HDR Jury

Rapporteurs:	Prof. Hubert Comon-Lundh	-	ENS Cachan
	Prof. Yassine Lakhnech	-	University Joseph Fourier
	Prof. John C. Mitchell	-	Standford University
Examinateurs:	Prof. David Basin	-	ETH Zurich
	Prof. Michael Hicks	-	University of Maryland
	Prof. Shriram Krishnamurthi	-	Brown University
	Prof. Bruno Martin	-	University of Nice-Sophia Antipolis

b

Contents

Acknowledgments

I've got a very enjoyable job and the opportunity to do this job in an area with tons of sunlight and great Nature. I was about to thank Inria for that but Inria is not a real person[1], so thanks to all the people that made my dream job[2] possible. My research is the result of a process full of interesting readings and inspiring discussions with other researchers. A special thanks to Andrei, Cédric, Frank, Gérard, Gilles, and all my other coauthors for that. Thanks to Zhengqin, José, Francis, Mihn, Mohammed and students of my courses belonging to the CASPAR UNSA master program: for exponentially accelerating my learning curve through constantly unexpected paths. Thanks to all the members that I proudly count on my HDR jury for kindly accepting to evaluate my work. Thanks also to the people who helped with all the administrative burden to make this HDR possible. Je suis très reconnaissante vers Manuel pas seulement pour son soutien mais AUSSI pour m' avoir introduite à des sujets de recherche très motivants.Thanks to Ana, Gilles, Marieke and all my family for being my biggest supporters, and specially thanks to Eze[3] for being a real partner [4] and offering me an irreplaceable support. Gracias a la vida, que me ha dado tanto [5] and thanks for all the coffee. The design of this document was inspired by the HDR [Nar14] of Francesco Zappa Nardelli.

Motivation

[1] See a discussion on reality and organizations in [Har17].
[2] My alternative dream job was to become a bartender in a Bar called Foo.
[3] I mean Eze the person and not EZE the airport in Argentina, which as many know, I also love.
[4] https://www.ted.com/talks/sheryl_sandberg_why_we_have_too_few_women_leaders
[5] https://www.youtube.com/watch?v=WyOJ-A5iv5I

A very short story and more

I defended my PhD on verification of information flow policies in November 2006. Since September 2006, I spent one year as expert engineer in the Microsoft Research-Inria Joint Center in Saclay and I have been employed as *chargée de recherche* at Inria Sophia Antipolis-Mediterranée afterwards, with roughly 1 year break due to maternity. This *mémoire d'habilitation* summarises [6] my 10 years of research from 2007 to 2017 and focuses on four research directions that I am considering for the forthcoming years. To be accessible to a wider public the presentation is intentionally non-technical, details are to be found in the published papers. The complete list of my publications since my PhD is listed at the end of the summary. In the text, citations to my own work between 2007 and 2017 appear in plain style as in [16] whereas citations to other works appear in alpha style as in [BSS11]. Software related to my research, which is listed before the bibliography, is also cited with the keyword "Tool" in front: although I have hardly written a line of code, I most often have participated to their design and/or their related security proofs. With the exception of Web Stats [Tool:Webstats], all the code represent prototypes to evaluate in practice the ideas proposed in their associated papers.

[6] The summary contains 42 pages. Any relation to 42 in [Ada78] is purely coincidental. Maybe.

Chapter 1

Programs using cryptography

In which we design type systems to check the correct usage of secure cryptographic schemes

The Problem Cryptographic algorithms can leak information about their encrypted payload[1] [Lip81], even if they rely on intractable problems, such as factoring or discrete logarithm. In the 80's, Goldwasser and Micali proposed the notion of semantic security [GM82], known today as indistinguishability (IND), a property that asserts that encryption cannot leak information to attackers with polynomial computational power. The proposal of semantic security was the first step towards the theoretical foundations of modern cryptography. Since then until today, many cryptographic algorithms have been proved to hold the IND security property in many of its declinations IND-CPA, IND-CCA, etc. However, in spite of these great advances in the state of the art, in practice, when programming with cryptography there might still be problems. If programs do not use cryptographic algorithms correctly, strong cryptographic theoretical guarantees are useless for the actual security of the payloads.

The Dream In an ideal world, practice will keep up with theoretical advances. For security in particular, this means that if certain cryptographic algorithms are proven to hold strong security properties to protect payloads, such as IND, then programs that use such algorithms will offer at least the same security guarantees for the encrypted payloads.

1.1 What can go wrong?

Cryptography provides essential mechanisms for confidentiality, with a wide range of algorithms [Mar04] reflecting different trade-offs between security, functionality, and

[1] The article [Lip81] turns out to be of crutial importance for people fonding for playing Poker.

efficiency. Thus, the secure usage of adequate algorithms for a particular system is far from trivial. Even with plain encryption, the confidentiality and integrity of keys, plaintexts, and ciphertexts are interdependent: encryption with untrusted keys is clearly dangerous, and plaintexts should never be more secret than their decryption keys. Integrity also matters: attackers may swap ciphertexts, and thus cause the declassification of the wrong data after their successful decryption. Conversely, to protect against chosen-ciphertext attacks, it may be necessary to authenticate ciphertexts, even when plaintexts are untrusted.

In the examples of the rest of the chapter, we use the following cryptographic schemes:

Encryption scheme: we use a (public key) encryption scheme $\langle \mathcal{G}_e, \mathcal{E}, \mathcal{D} \rangle$, that is a triple of algorithms such that $\mathcal{G}_e$ is a probabilistic key generation algorithm, $\mathcal{E}$ is a public key encryption probabilistic algorithm, and $\mathcal{D}$ is its correspondent decryption deterministic algorithm.

We assume the scheme to be correct, that is, decryption of an encryption is the identity function for a plaintext element given that the encryption and decryption keys were generated by a unique call to $\mathcal{G}_e$.

We assume the scheme to hold the security property of Indistinguishability against Chosen Chiphertext Attacks, a.k.a. IND-CCA [RS91]. Such an indistinguishability property states that if an adversary[2] is given the opportunity to decrypt a set of ciphertexts that he has chosen to encrypt, he will still get **no information**[3] from ciphertexts that are not in that set. (In this mémoire we keep the algorithms of the encryption scheme abstract. A notable instantiation of the encryption scheme used in this section is OAEP-RSA, which was proved IND-CCA in [FOPS01].)

Signature scheme: we use a signature scheme $(\mathcal{G}_s, \mathcal{S}, \mathcal{V})$, that is a triple of algorithms such that $\mathcal{G}_s$ is a probabilistic key generation algorithm (generating a private key k_s for signatures, and a public key k_v for verification), $\mathcal{S}$ is a probabilistic signing algorithm, $\mathcal{V}$ is a deterministic signature verification algorithm.

We assume that the signing scheme is correct, that is verification of a signature of a message returns true, $\mathcal{V}(m, \mathcal{S}(m, k_s), k_v) = true$, for all $k_s, k_v := \mathcal{G}_s()$ and message m in the plaintexts set.

We assume that the signing scheme is secure against forgery under adaptive Chosen Message Attacks a.k.a. CMA [GMR88]. Such a security property states that even if an adversary is given the opportunity to sign a set of chosen messages, he will **not be able to forge any signature**[4] of a message outside that set.

[2] An adversary in this context is a program which is polynomial on the length of the key.

[3] Except for a negligible probability.

[4] Except for a negligible probability.

Blinding scheme: we use a blinding scheme $(\mathcal{G}_e^B, \mathcal{P}, \mathcal{B}, \mathcal{D}^B)$ such that $(\mathcal{G}_e^B, \mathcal{P}; \mathcal{B}, \mathcal{D}^B)$ is an encryption scheme, where $\mathcal{P}; \mathcal{B}$ is the composition of two algorithms:

- pre-encryption $\mathcal{P}()$ inputs a plaintext and outputs its representation as a ciphertext, but does not in itself provides confidentiality; it is deterministic;
- blinding $\mathcal{B}()$ operates on ciphertexts; it hides the correlation between its input and its output, by randomly sampling another ciphertext that decrypts to the same plaintext.

We assume that the blinding scheme is correct, that is, the encryption scheme it defines $(\mathcal{G}_e^B, \mathcal{P}; \mathcal{B}, \mathcal{D}^B)$ is correct and $\mathcal{B}$ is a probabilistic function such that, for all $k_e, k_d := \mathcal{G}_e^B()$, if v encrypts m, then $\mathcal{D}^B(v, k_d) = m$, where 'encrypts' is defined by

1. v encrypts m when $v := \mathcal{B}(\mathcal{P}(m, k_e))$ with m a plaintext;
2. v' encrypts m when $v' := \mathcal{B}(v, k_e)$ and v encrypts m.

We assume the blinding scheme to hold the security property of Indistinguishability against Chosen Plaintext Attacks a.k.a. IND-CPA. This property provides the same security guarantees than IND-CCA but against a weaker adversary which does not have the opportunity to decrypt chosen ciphertexts.

Homomorphic encryption scheme: An homomorphic encryption scheme is an encryption scheme that permits homomorphic operations on ciphertexts. We use an homomorphic encryption scheme $\langle \mathcal{G}_e^H, \mathcal{E}^H, \mathcal{D}^H \rangle$ that permits to add plaintexts by multiplying their ciphertexts (an instantiation of such scheme is Paillier [Pai99]). We assume the scheme is correct and IND-CPA.

The following is a short and incomplete tour of programs that invalidate confidentiality or integrity even when using provably strong cryptographic schemes.

1.1.1 Key cycles

IND provides security guarantees if the adversary does not hold the decryption key. Hence, in particular, IND-CCA and IND-CPA do not provide any guarantee if the adversary gets the decryption key as plaintext nor in more complex usages of encryptions, such as those where ciphertexts are encryptions of decryption keys. Said otherwise, IND-CCA and IND-CPA say nothing about confidentiality in case the plaintexts may depend on the decryption key (see e.g.[AR02]). This situation is referred to as a key cycle. Programs which use IND-CCA or IND-CPA schemes but present key cycles, loose the advantages offered by strong security guarantees (other

security properties [BRS02, ABHS09] offer security guarantees even in the presence of key cycles).

Example 1 (Key cycle)**.** *Consider the program P defined as:*

$$\begin{aligned} P \doteq\ & k_e, k_d := \mathcal{G}_e(); \\ & k'_e, k'_d := \mathcal{G}_e(); \\ & x := \mathcal{E}(k'_d, k_e); \\ & y := x; \\ & x' := \mathcal{E}(y, k'_e) \end{aligned}$$

The program P has a key cycle. One may have that even if $\langle \mathcal{G}_e, \mathcal{D}, \mathcal{E} \rangle$ is IND-CCA the encryption in x' leaks information on key k'_d.

1.1.2 Wrong decryption keys

Correctness of encryption schemes is guaranteed only if the encryption and decryption keys match, that is, they are generated by a single call to $\mathcal{G}_e$.

Example 2 (Bad decrypted key)**.** *Consider the following program where s is a secret :*

$$\begin{aligned} & k_e, k_d := \mathcal{G}_e(); \\ & k'_e, k'_d := \mathcal{G}_e(); \\ & x_{LL} := \mathcal{E}(k'_e, k_e); \\ & k_d := k'_d; \\ & k := \mathcal{D}(x_{LL}, k_d); \\ & y := \mathcal{E}(s, k) \end{aligned}$$

This program is insecure because the key k_d used for decryption does not match the key k_e used for encryption. Hence, the decrypted value k is unspecified—it is unlikely to be a valid encryption key—and encryption using k is also unspecified—one may have then that even if $\langle \mathcal{G}_e, \mathcal{D}, \mathcal{E} \rangle$ is IND-CCA the final encryption leaks both its parameters s and k.

1.1.3 Integrity of signing keys

A signing scheme offers integrity guarantees only if the signing key cannot be manipulated by the adversary.

Example 3 (Low-integrity signing keys)**.** *Consider a distributed program with two components: P_1 executes in one machine and P_2 in another. Assume that P_2 has a (correctly generated) pair of encryption keys k_e and k_d such that k_e is public (so*

that P_1 and the adversary also have it). Assume that P_1 has a (correctly generated) pair of signing/verification keys k_s and k_v such that k_v is public (so that P_2 and the adversary also have it). We abstract away details of communication, since they are unnecessary for what this example illustrates. Instead, we assume that variables with subindex LL are sent to the network.

$$P_1 \doteq e_{LL} := \mathcal{E}(k_s, k_e)$$

$$\begin{aligned} P_2 \doteq & \\ & \mathsf{if}\ s\ \mathsf{then}\ k := \mathcal{D}(e_{LL}, k_d); \\ & z_{LH} := 0; \\ & m_{LL} := S(z_{LH}, k) \end{aligned}$$

If an adversary sets e_{LL} before P_2 receives it, and then tests whether m_{LL} verifies, he can infer whether the secret s is true.

1.1.4 Replay attacks

CMA security does not require security against replay: an adversary may use an old and faithfully generated (by a trusted party) signature in order to "forge" a new signature. Thus the integrity of a message can be violated even when using CMA signing schemes, as shown in the following example.

Example 4 (Replay attack)**.** *Consider a distributed program with components P_1 and P_2.*

$$\begin{aligned} P_1 \doteq\ & x := 0; x' := 1; \\ & y_{LL} := \mathcal{S}(x, k_s); z_{LL} := x; \\ & y'_{LL} := \mathcal{S}(x', k_s); z'_{LL} := x' \end{aligned}$$

$$P_2 \doteq \mathsf{if}\ \mathcal{V}(z_{LL}, y_{LL}, k_v)\ \mathsf{then}\ h_{LH} := z_{LL}$$

An adversary that intercepts the messages between P_1 and P_2 may execute the following code: $y_{LL} := y'_{LL}; x_{LL} := x'_{LL}$ causing a violation in the integrity of the message, since P_2 believes that P_1 sent 1 instead of 0.

1.1.5 Missing blinding

Example 5. *Consider a program choosing between two ciphertexts depending on a secret value s:*

$$y_{LH0} := \mathcal{E}(x_{HH0}, k_e);$$
$$y_{LH1} := \mathcal{E}(x_{HH1}, k_e);$$
$$\mathsf{if}\ s\ \mathsf{then}\ y := y_{LH0}\ \mathsf{else}\ y := y_{LH1}$$

If the adversary gets y,y_{LH0}, y_{LH1}, he may compare y with y_{LH0} and y_{LH1} and infer the value of s. The program would be secure with the addition of one line of code using blinding $y := \mathcal{B}(y, k_e)$ to randomise the original encryption.

1.1.6 Honest but curious adversaries

Example 6. *Consider the following program that homomorphically multiply an encrypted value by a small integer factor.*

$$k_e, k_d := \mathcal{G}_e^H();$$
$$y := \mathcal{E}^H(x_{HH}, k_e);$$
$$z := \mathcal{E}^H(2, k_e);$$
$$\mathsf{for}\ i := 1\ \mathsf{to}\ n\ \mathsf{do}\ z := z * y$$

An adversary with access to z and y may also iterate multiplications of y and z and compare them to the result to guess n. □

1.2 Typable programs do not go wrong

Cryptographic scheme features are naturally explained in terms of information flows; they enable computations at a lower level of confidentiality, for example homomorphic operations can be delegated to an "honest but curious principal" but they require some care in the presence of active adversaries.

We have developed information flow type systems [5, 13] and safety conditions for checking the correct usage of cryptography. All examples of Section 1.1 are rejected by our type systems.

The typing rules reflect the diverse functional and security features of encryption and signing schemes and allow us to prove that programs using cryptography preserve confidentiality and integrity properties of sensitive data, under strong cryptographic assumptions such as IND-CPA, IND-CCA, and CMA. In order to mathematically define the security properties of confidentiality and integrity that we aimed at achieving, we have proposed the notion of Computational Noninterference against active adversaries (CNI). CNI is closely related to the notion of semantic security in cryptography but applies to (probabilistic) programs using cryptography instead

of to cryptographic schemes. CNI states that programs cannot leak[5] information to attackers with only polynomial computational power.

To assert that a program complies with CNI, it is necessary to distinguish what is public from what is secret. For that, we propose to use information flow specifications, where security levels, belonging to a given lattice, indicate the sensitivity of the information. Thus, in this setting, we assume that adversaries can read program resources labeled up to a certain (confidentiality) label, and write resources down to a certain (integrity) level. The definition considers active adversaries: adversaries that can interact with our trusted programs. We model adversaries as polynomial-bounded probabilistic programs and the locations where they can interact with trusted programs are modeled as holes in program contexts.

The type system developed in [5], and extended in [13], uses the following types for cryptographic values:

$\tau ::=$	$t(\ell)$	Security types
$t ::=$	Data	Data types for payloads
$\mid$	$\mathsf{Enc}\,\tau\,K \mid \mathsf{Ke}\,\tau\,K \mid \mathsf{Kd}\,\tau\,K$	Data types for encryption
$\mid$	$\mathsf{Sig}\,\tau \mid \mathsf{Ks}\,F\,K \mid \mathsf{Kv}\,F\,K$	Data types for signing

where ℓ is a security label from a lattice of security levels, τ is a security type, F is a map from tags to security types, and K is a key label, as explained below.

Static key labels The labels K are used to keep track of keys, grouped by their key-generation commands. These labels are attached to the types of the generated key pairs, and propagated to the types of any derived cryptographic materials. They are used to match the usage of key pairs (see Example 2), to prevent key cycles (see Example 1), and to prevent generating multiple signatures with the same key and tag. Technically, we rely on τ in $\mathsf{Sig}\,\tau$ and on τ in $\mathsf{Ke}\ \tau\,K$ and $\mathsf{Kd}\,\tau\,K$ only to exclude the creation of encryption cycles. We also impose for soundness that each key label is used in at most one (dynamic) key generation.

Example 7 (Program context with a potential key cycle)**.** *Consider the following program context:*

$$\begin{aligned} P[\_] \doteq\ & k_e, k_d := \mathcal{G}_e();\\ & k'_e, k'_d := \mathcal{G}_e();\\ & x := \mathcal{E}(k'_d, k_e);\\ & \_;\\ & x' := \mathcal{E}(y, k'_e) \end{aligned}$$

[5]Except for negligible probability.

where the hole in the forth line represents a place holder for adversarial code. If variable y can be written by the adversary and variable x can be read by him, then the program $P[y := x]$ has a key cycle (and matches the program of Example 1), and thus the program context $P[\_]$ has a potential key cycle. If we assume that the integrity of x is such that the adversary can write on it, then to type this program, it is not possible to directly encrypt k'_d with k_e and at the same time encrypt k_d with k'_e since in our type system the type of variable x must contain the type of its payload - key k'_d - and the type of variable x', the type of encryption with key k_e , syntactically, we cannot write a type that is cyclic.

Encryption $\mathsf{Enc}\,\tau\,K$ represents an encryption of a plaintext with security type τ; $\mathsf{Ke}\,\tau\,K$ and $\mathsf{Kd}\,\tau\,K$ represent encryption and decryption keys, respectively, where τ indicates the security type of plaintexts that may be encrypted and decrypted with these keys. We generalise this to a set in [13] to enable typechecking for code that uses the same key pair for different purposes, which is important for efficiency.

Tagged signatures $\mathsf{Sig}\,\tau$ represents an (unverified) signature on a value with security type τ. $\mathsf{Ks}\,F\,K$ and $\mathsf{Kv}\,F\,K$ represent signing and verification keys, respectively. Cryptographic signatures are often computed on (hashed) texts prefixed by a tag t or some other descriptor that specializes the usage of the signing key to avoid replay attacks. Accordingly, in order to precisely type expressions of the form $\mathcal{S}(\mathsf{t}+m, k_s)$ where t is a constant tag and m is the signed text, our key types embed a partial map, F, from the tags usable with the key to the security types of the corresponding signed values. We assume that the tags in the domain of F all have the same size. Without this limited form of type dependency, we would essentially have to use a distinct key for every signature. We impose that each signing-key label/tag pair is used for signing at most once to prevent replay attacks as the one shown in Example 4.

Example 8 (Tagged signatures)**.** *The command context*

$$\begin{aligned} P[\_] \doteq\ & k_s, k_v := \mathcal{G}_s();\\ & y_{LL} := \mathcal{S}(\mathsf{t0} + x_{LH}, k_s); z_{LL} := x_{LH};\\ & y'_{LL} := \mathcal{S}(\mathsf{t1} + x'_{LH}, k_s); z'_{LL} := x'_{LH};\\ & \_;\\ & \mathsf{if}\ \mathcal{V}(\mathsf{t0} + z_{LL}, y_{LL}, k_v)\ \mathsf{then}\ h_{LH} := z_{LL} \end{aligned}$$

is CNI (and typable) against an adversary that can read and write y_{LL}, y'_{LL}, z_{LL}, z'_{LL}. If we assume that x_{LH} and x'_{LH} have type $\mathsf{Data}\,(LH)$, we may use, for instance, a map F defined by $F(\mathsf{t0}) = F(\mathsf{t1}) = \mathsf{Data}\,(LH)$.

Applications We illustrate the effectiveness of our type systems using programming examples based on key establishment protocols [5] and classic encryption schemes with homomorphic properties [ElG84, Pai99]. We also develop two challenging applications. Both applications rely on a security lattice with intermediate levels, reflecting the structure of their homomorphic operations, and enabling us to prove confidentiality properties, both for honest-but-curious servers (see Example 6) and for compromised servers controlled by an active attacker.

- We program and typecheck a practical protocol for private search on data streams proposed by Ostrovsky and Skeith III [OSI05] , based on a Paillier encryption of the search query. This illustrates that our types protect against both explicit and implicit information flows; for instance we crucially need to apply some blinding operations to hide information about secret loop indexes.
- We program and typecheck the bootstrapping part of Gentry's fully homomorphic encryption [Gen09]. Starting from the properties of the bootstrappable algorithms given by Gentry – being CPA and homomorphic for its own decryption and for some basic operations –we obtain an homomorphic encryption scheme for an arbitrary function. This illustrates three important features of our type system: the ability to encrypt decryption keys (which is also important for typing key establishment protocols); the use of CPA encryption despite some chosen-ciphertext attacks; and an interesting instance of homomorphism where the homomorphic function is itself a decryption.

Our type systems exclusively treat public-key encryptions. We believe that their symmetric-key counterparts can be treated similarly.

Chapter 2

Programs in the World "Wild" Web

In which we discuss essential sources of web vulnerabilities and some countermeasures

Problem Today's web applications are a mix of existing online libraries and data that are combined to write applications in a rapid and inexpensive manner. However, this same flexibility together with the mix of heterogeneous technologies make the task of programming secure web applications very complex. As web application vulnerabilities are becoming widespread, web application developers are facing an unprecedented need of tools to help smooth handling web vulnerabilities.

The Dream The web is a gigantic source of libraries and ready to use functionalities. The dream is to keep this without the nightmare of vulnerabilities that come along it.

2.1 When the good becomes the evil

Web applications are distributed programs that share intrinsic characteristics such as the server-client dichotomy, client execution in a browser, and specific programming languages and internet protocols. In the following, we show characteristics of web applications that distinguish them from other distributed programs and that, in my opinion, are the essential source of the most popular web vulnerabilities.

2.1.1 Browsing the web, a dynamic experience

In contrast to other distributed applications, the model of execution of web applications does not start with different code installed in different nodes. Actually, the

whole code to be executed as part of the distributed application does not necessarily even exist when the web application execution starts. Rather, only some of the nodes (servers) contain code that dynamically generate the code for other nodes (clients). When a client starts the execution of a web application from the browser, the server "computes" the code which is to be sent to the client. This server computation may use input data provided by the client to start the execution. This particular characteristic of web applications -client code generated on-the-fly- represents a real advantage to dynamically integrate, for example, data from a database into a web page according to the selection of a client. This single characteristic highly improves user's experience and, for example, makes e-commerce applications possible. At the same time, this single characteristic is enough for the existence of code injection attacks. Code injections attacks (a.k.a. XSS attacks when the injection occurs on the client) occur when the input data provided by the client is directly used by the server to generate the client page (resp. a database query), without or with a wrong sanitization. For the server, the input data is data but, if the input is malicious, this data can be interpreted as code in the client (resp. as a query for the database management system).

Example 9 (Code injection attack)**.** *The following server code is vulnerable to XSS because the input name provided by the client is not sanitized:*

```
<html>
   <body>
       Welcome <?php echo $_GET[name] ?>
   </body>
</html>
```

If the server code is accessible via `http://www.example.com`*, then a malicious client can use the following attack vector:* `http://www.example.com?name='<script>alert(Hacked!)</script>'` *and his script will be executed in the client browser. This is the simplest and least dangerous form of XSS: in other forms, the attacker can permanently store the attack vector to simultaneously attack all the vulnerable server's clients.*

Code injection attacks (XSS, server, and other web injection attacks confused) have consistently make it to the top of the most popular web vulnerabilities, such as OWASP [OWA] top ten, through the years.

2.1.2 A giant source of resources

Since 2005, web applications have offered the potential to finally make widespread software reuse a reality. Most of today web clients integrate code generated from one main server and third-party code from several other servers. Client code generated in

such a way is also known as a mashup. Code coming from the main server is called the mashup integrator. Third-party code, often called gadget, may be untrusted (since it is not generated by the main server). Third-party code in a mashup can be included in two ways: either by using the script tag and granting access to all the resources of the integrator, permitting to execute untrusted code with otherwise impossible integrator privileges; or by using the iframe tag, in which case the gadget is isolated.

Example 10 (Mashup built with the script tag)**.** *In the following mashup code, the second script belongs to a third-party and untrusted server. Still, the gadget script has been included with full privilege to access all webpage's resources.*

```
<html>
  <body>
     <div id=gadget_canvas></div>
     <script src="trusted.js"></script>
     <script src="http://attacker.com/adservice.js"</script>
   </body>
</html>
```

The integrator calls methods or functions as interfaces to change the state of the gadget. For example, the following is a code snippet (in the integrator script trusted.js) to manipulate the untrusted gadget via interfaces:

```
var mydiv = document.getElementById("gadget_canvas")
var instance = new gadget.newInstance(
                      mydiv, gadget.Type.SIMPLE);
instance.setLevel(9);
```

The gadget defines a global variable `gadget` *to provide interfaces to the integrator. The* `gadget.newInstance` *is used to create a new gadget instance that binds to the* `div`*; and* `instance.setLevel` *is a method used to change state at the gadget instance. Let us assume that the integrator stores a secret in global variable* `secret` *and a global variable* `price` *holding certain information with an important integrity requirement:*

```
var secret = document.getElementById(secret_input());
var price =  42;
```

The secret flows to an untrusted source, and the price is modified at the gadget's will if the gadget code adservice.js contains the following code:

```
var steal;
steal = secret;
price = 0;
```

If the gadget adservice.js is isolated using the iframe tag with a different origin, variables `secret` *and* `price` *cannot be directly accessed by the gadget. However, by using the iframe tag, the interface of the gadget for the integrator, object* `gadget`*, cannot be directly accessed neither. This poses a problem for the functionality of the mashup.*

Mashup programmers are challenged to provide flexible functionality even if the code consumer is not willing to trust third-party code that mashups utilize. Unfortunately, programmers often choose to include third-party code using the script tag and resign to security in the name of functionality.

2.1.3 Infinite clients, or almost

HTTP is the most widely used communication protocol in web applications. One important detail regarding HTTP is that it is stateless: the server does not retain any information or status after responding to an HTTP request. This stateless design has been historically important for scalability[1] : it allows the server to handle a high number of client requests without the need to dynamically allocate storage for each conversation. The stateless nature of HTTP imposes other ways to keep state in conversations between a client and a server, a.k.a. sessions. Currently, a common way to add state in sessions is the use of cookies which are permanently stored in the client while the session is active. Cookies are usually created by the server, and starting from their creation, cookies are automatically sent from the client to the server with each HTTP request. Whenever the server receives a cookie, it learns information on the session, including for example if the user has correctly logged in in the past. The server learns that a session with that client is active. Fully relying on cookies to implement sessions and storing them only on the client opens up a single point of failure in order to attack a web session (e.g. change the sense of conversations between client and servers). Not surprisingly, cookies are often the target of web applications attacks, such as XSS, which often aim at stealing the (session) cookies.

Example 11 (Cookie stealing attack vector)**.** *An attacker who is able to inject code or who can rightfully execute code because his code is included with the script tag, can steal cookies via the following line of code:*

```
window.location="http://attacker.com?cookie="+document.cookie;
```

The browser process of automatically sending the cookie to the server is also the target of attacks, such as cross-site request forgery or XSRF, which aim at using the power of a session cookie without even stealing it.

[1] It is unclear that the stateless design remains important for scalability today.

Example 12 (XSRF attack)**.** *If a user is logged in a service vulnerable to XSRF and he visits a site* `http://attacker.com` *from the attacker with the following code*

```
<html>
 <body>
   <img src="http://vulnerable.com/MyAccount?NewState=Hacked"
      width="1" height="1"  />
 </body>
</html>
```

(Alternatively, the attacker may inject the above `img` *tag into the user's webpage.) When parsing the img tag, the browser tries to load an image (that doesn't need to exist and that remains invisible to the user since it is a 1 pixel image) from the vulnerable server, sending an HTTP request to it. Since the browser automatically sends the session cookie with the HTTP request to keep the state, the server interprets that the request comes from an authentic user (instead of coming from* `http://attacker.com`*) willing to change the state of his account.*

2.1.4 JavaScript: programming for the masses

JavaScript (which was officially called LiveScript in the beginning and it is now officially called ECMAScript [ECM]) was designed to allow non expert programmers to quickly write code to assemble different components in web pages. It was created in 10 days[2]. Since then, JavaScript has become the most widely used programming language for client web applications and, currently, it is one of the only two languages natively supported by all major web browsers. At first sight, JavaScript code looks simple and intuitive. However JavaScript is a complex programming language with a large specification full of corner cases [ECM, MMT08, GSK10, BCF+14] and often written in an event-driven style [KHFS09, BS14]. This complexity has misled programmers to unintentionally develop vulnerable code. We show an example [3] that was the base of the attack to Facebook reported in [MMT09].

Example 13 (Insecure JavaScript code that seems secure)**.** *The following function is meant to forbid the access to property "secret" of the window global object:*

```
function accessControl(prop){
  if (prop==="secret")
    { return "Access forbidden"; }
  else {
      return window[prop]; }
 }
```

[2]Source: https://en.wikipedia.org/wiki/JavaScript

[3]This example was inspired by a talk of Shridam Krishnamurthi in 2011, Sophia Antipolis.

If this access control function is called for each property access, does this function make its job to forbid the access to `secret`*? The answer is* **No**. *In JavaScript, object property names must be strings but they can be computed dynamically. For instance, one can inspect the content of a property p of an object o by writing o[e] where e is an expression that dynamically evaluates to p. Thus, a possible attack vector (an input prop provided with the goal of accessing "secret" without authorization) is the following object defined in variable attack:*

```
attack = {toString:function(){return "secret"}}
```

The attack vector defines an object with a property called `toString`*. It turns out that JavaScript also possesses a native function called* `toSring` *and that, as stated in the ECMAScript specification, this function is implicitly invoked whenever a value, as parameter of an access, is not a string. In a call to* `accessControl(attack)`*, the conditional evaluates to false because* `attack` *is not of the same type as the string* `"secret"` *(*`attack` *is of a different type, object). Hence, the access* `window[attack]` *evaluates to* `window["secret"]`*:* `attack` *is first subjected to an implicit call to* `toString`*. According to JavaScript scoping rules, the* `toString` *function that executes is the one defined in the* `attack` *object, which finally evaluates to* `"secret"`*.*

2.2 Noninterference for JavaScript and web APIs

Some popular attacks on the web can be modeled as an instance of the more general problem of information flow [JJLS10]. For example if a cookie, which is considered confidential, is sent to (or flows to) a web server considered malicious, the cookie stealing attack (see Sect.2.1.3) is caused by an insecure information flow, which is not controled, in the web application.

Having long been studied in the literature [SM03, SS09], information flow policies provide the mathematical foundation for reasoning precisely about the information flows that take place during the execution of a program.

Information flow policies can be statically or dynamically enforced. Currently, five main techniques have been proposed for dynamic information flow enforcement: no-sensitive-upgrade [Zda02], permissive-upgrade [AF10], hybrid monitors [LGBJS06], secure multi-execution [DP10], and multiple facets [AF12].

We have evaluated the applicability of the no-sensitive-upgrade technique to real-world web client-side code: we have proposed the first compiler [Tool:Iflowsig] that instruments JavaScript code to inline an information flow monitor based on the no-sensitive-upgrade technique. The compiler is defined for a (large) subset of JavaScript [19] and the DOM API [18], and it can be modularly extended to other APIs [21]. The compiler is sound with respect to the information flow policy of noninterference, a policy that strictly forbids any flow of information between

resources belonging to different classes of confidentiality classifications. For example, if a cookie is classified as confidential, the execution of a compiled program will only go through if the cookie is never sent to a server without the clearance to see confidential data; otherwise, the constraints inlined in the program by the compiler will cause it to diverge.

We have recently built a taxonomy of the main dynamic information flow enforcement techniques according to the properties of transparency and soundness [24, 25], concluding that secure multi-execution and multiple facets, and hybrids of those [NFR+18], are the techniques with greater transparency and stronger guarantees.

On one hand, the use of purely dynamic mechanisms comes at the cost of a much lower performance compared to the original code [DDNP12]. On the other hand, the dynamic aspects of JavaScript constrain the use of purely static analyses to language subsets (that exclude commonly used features) or to over-approximations with an unacceptable (to be applicable in practice) precision loss.

Example 14 (Sources of JavaScript static analyses imprecision)**.** *The dynamic computation of property names (see Example 13) is one of the major sources of imprecision of static analyses for JavaScript [MT09]. To better understand why that is the case, consider the following program:*

```
o = {};
o.secret = secret_input();
o.public = public_input();
public =  o[f()]
```

This program creates an object `o` *that has a secret property* `secret` *assigned to a secret input obtained via the function* `secret_input`*, and a public property* `public` *assigned to public inputs obtained through the function* `public_input`*. The program assigns the value of one of the properties o the public variable* `public`*. The actual property whose value is assigned to* `public` *is determined by the return value of* `f`*. When* `f` *returns the string* `"secret"`*, the program assigns a secret value to* `public`*, and the execution is insecure. On the other hand, when* `f` *returns jsinline"public", the program assigns a public value to* `public` *and the execution is secure. However, it may be the case that* `f` *never returns* `"secret"`*. If this is so, the execution is always secure and, therefore, the program can be considered secure. But, in order to detect this case, an analysis would have to predict the dynamic behaviour of the function* `f`*, which is, in general, undecidable.*

We have proposed a general hybrid analysis to statically verify secure information flow in JavaScript. Following the hybrid typing motto "static analysis where possible with dynamic checks where necessary" [Fla06], we are able to reduce the runtime overhead introduced by purely dynamic analyses without excluding dynamic field

operations. In fact, our analysis can handle some of the most challenging JavaScript features, such as prototype-based inheritance, extensible objects, and constructs for checking the existence of object properties. Its key ingredient is an internal boundary statement inspired by work in inter-language interoperability [MF09]. The static component of our analysis wraps program regions that cannot be precisely verified inside an internal boundary statement instead of rejecting the whole program. This boundary statement identifies the regions of the program that must be verified at runtime -which may be as small as a single statement- and enables the initial set up required by the dynamic analysis. In summary, the proposed boundary statement allows the semantics to effortlessly interleave the execution of statically verified code with the execution of code that must be verified at runtime.

Previous analyses for enforcing security properties in JavaScript, such as that of [MT09], have chosen to restrict the targeted language subset, excluding dynamic field operations. In contrast, we account for the use of these operations by verifying them at runtime. When verifying a statement containing a dynamic field operation, the static analysis wraps it inside a boundary statement. For instance, in the program of Example 14, the last assignment is re-written as

$$@monitor(@type\_env, @pc, @ret, public = o[f()])$$

where the first three arguments of the monitor statement are used for the setup of the runtime analysis. All the other statements, which do not contain dynamic field operations, are fully statically verified for noninterference according to the programmer specified security policy[4] and, therefore, left unchanged.

A prototype of the proposed analysis is available online [Tool:Iflowtypes].

2.3 Browser security

Modern browsers implement different specifications to securely fetch and integrate content. One widely used specification to protect content is the Same Origin Policy (SOP) [SOP]. SOP allows developers to isolate untrusted content from a different origin. If an iframe's content is loaded from a different origin, SOP controls the access to the embedder resources (see Section 2.1.2). In particular, no script inside the iframe can access content of the embedder page. However, if the iframe's content is loaded from the same origin as the embedder page, there are no privilege restrictions w.r.t. the embedder resources. In such a case, a script executing inside the iframe can access content of the embedder webpage. Scripts are considered trusted and the

[4] As property names can be dynamically computed, it is not realistic to expect the programmer to always know which properties are going to be dynamically created. To account for this issue, we introduced security types that allow for the policy specification of a *default security type* [Thi05] for all the properties of an object not known *a priori*.

iframe becomes transparent from a developer view point. A more recent specification to protect content in webpages is the Content Security Policy (CSP) [SSM10]. The primary goal of CSP is to mitigate XSS attacks. CSP allows developers to specify, among other features, trusted domain sources from which to fetch content. One of the most important features of CSP, is to allow a web application developer to whitelist trusted JavaScript sources. This kind of restriction is meant to permit execution of only trusted code and thus prevent untrusted code to access content of the page.

We have reported on a problem with CSP [30] and its interplay with the SOP policy: CSP does not take into consideration the page's context, that is its embedder or embedded iframes. In particular, CSP is unable to protect content of its corresponding page if the page embeds (using the *src* attribute) an iframe of the same origin. The CSP policy of a page will not be applied to an embedded iframe. However, due to SOP, the iframe has complete access to the content of its embedder. We analysed 1 million pages from the top 10,000 Alexa sites and found that 5.29% of sites contain some pages with CSPs.We have identified that in 94% of cases, CSP may be violated in presence of the document.domain API and in 23.5% of cases CSP may be violated without any assumptions. During our study, we also identified a divergence among browsers implementations in the enforcement of CSP [WBV15] in sandboxed iframes embedded with *srcdoc*, which actually reveals an inconsistency between the CSP and HTML5 sandbox attribute specification for iframes. (We have made publicly available the dataset that we used for our results in `http://webstats.inria.fr/?cspviolations`. We have installed an automatic crawler to recover the same dataset every month to repeat the experiment taking into account the time variable.) Other fundamental problems have been reported with CSP [Joh14, SCB16, WSLJ16]: perhaps the most surprising of these problems is that the whitelisting mechanism of CSP simply doesn't work [WSLJ16] since real-world deployments result in bypasses in 94.72 % of all distinct CSP policies.

More effective solutions for browser security have been proposed[5].

Secure Multi-Execution (SME) [DP10], one of the dynamic mechanisms mentioned in Section 2.2 [24, 25, NFR+18], is a precise and general information flow control mechanism that was implemented in FlowFox [DDNP12], a browser that extends Mozilla Firefox. FlowFox supports powerful security policies without compromising compatibility. SME multi executes programs, in a blackbox manner, as many times as security levels in a lattice. The essence of the SME mechanism [DP10] can be described by the following (extremely simplified) rule:

[5]Unfortunately, in an ongoing study, we notice that they [WSLJ16, SCB16, CRB17] are not yet widely adopted in practice.

$$\frac{(P, [l : vl, h : vh]) \Downarrow [l : vl_H, h : vh_H] \qquad (P, [l : vl, h : 0]) \Downarrow [l : vl_L, h : vh_L]}{(P, [l : vl, h : vh]) \Downarrow_{SME} [l : vl_L, h : vh_H]}$$

In this rule, P is a program, in some language with a semantics given by the relation $\Downarrow$, and $[l : vl, h : vh]$ is an initial memory which maps a public variable l to value vl and a secret variable h to value vh. We assume only two confidentiality levels defining public and secret views. In order to evaluate P in the SME semantics $\Downarrow_{SME}$, the program is executed twice. The first execution corresponds to the normal execution of the program using its own semantics $\Downarrow$ and initial memory $[l : vl, h : vh]$. It computes the secret view in variable h. The second execution is modified: it also uses the normal semantics but the value of the secret variable h has been modified to be the dummy value 0. It computes the public view in variable l. Finally, the result of the SME evaluation is a memory where variable h, corresponding to the secret view, is assigned to the value vh_H of the normal execution, and variable l, corresponding to the public view, is assigned to the value vl_L of the modified execution. Since the value offered to the public view is computed without any secrets (secret h has been reset to a dummy value 0), program P is trivially noninterferent. (Notice that the simplified rule does not take termination into account. The original SME paper is shown to enforce a stronger property considering termination. However, in a recent paper [NPR18], we identify some inconsistencies regarding termination in the original SME soundness theorem.)

Example 15 (SME in practice)**.** *The following JavaScript code implements a simple key logger. It installs an event handler to monitor key presses, and leaks every keystroke to* `hacker.com`*. It does this by encoding the character code of the key in an image URL and asking the browser to fetch that URL.*

```
var url = 'http://attacker.com/?=';
window.onkeypress = function(e) {
  var leak = e.charCode;
  new Image().src = url + leak;
}
```

Potential leaks by running this code are prevented in FlowFox: by means of the SME mechanism only dummy values will be sent to `attacker.com`*.*

Unfortunately, strict noninterference breaks some functionality that is important for the web today.

Example 16 (Need for more relaxed security policies)**.** *Consider the following simple variant of web analytics: a web-based application that wants to analyse which keyboard shortcuts are commonly used. It is common practice on the web to include*

third party analytics scripts to gather such information. For this use case, there is no need to know in what order keys were pressed nor how many times a particular key was pressed, only whether a certain key was pressed at least once during the interaction with the web application. Releasing such limited information poses negligible security risks, and can be considered acceptable and even useful in many situations. The following script sends this minimal amount of information to `analytic.com`*. This is a simple example of a very broad set of practices on the web today, where third party web analytics companies monitor application usage and gather statistics (such as mouse heat maps or the geographical spread of users).*

```
var d = 0, url = 'http://analytic.com/?=';
window.onkeypress = function(e) {
  if (e.charCode == 101) d = 1; }
window.onunload = function() {
  $.ajax(url + d); }
```

Another example where strict information flow control breaks functionality is when the labelling of incoming information is too coarse grained. For instance, when cookies are marked as confidential to defend against XSS (see Example 9), also non-sensitive information stored in cookies (such as the preferred language) is no longer accessible to low observers, and this can break script functionality.

The original SME mechanism does not distinguish between the malicious key logger from Example 15 and the useful benign script from Example 16: both scripts leak private information (key presses) to network servers. It does neither support a fine grained approach to label incoming event information: events are either high or low. What is needed to support such scenarios is some form of declassification: a policy should specify what kind of aggregate, derived, or partial information is safe to release to low observers. We have proposed stateful declassification policies [20] for event-driven programs such as JavaScript applications, and developed an enforcement mechanism for these policies on top of the existing SME mechanism [DP10, DDNP12]. Our declassification policies are based on relaxed noninterference, a particularly expressive approach proposed by Li and Zdancewic [LZ05]. This kind of declassification enforces a notion of relaxed noninterference by allowing programmers to specify policies that capture the intended manner in which public information can be computed from private data.

The essence of the declassification SME mechanism [20] can be described by the following (extremely simplified) rule:

$$\frac{\begin{array}{c}(P, [l : vl, h : vh]) \Downarrow [l : vl_H, h : vh_H] \\ v = \texttt{declassify}(vh) \qquad (P, [l : vl, h : v]) \Downarrow [l : vl_L, h : vh_H]\end{array}}{(P, [l : vl, h : vh]) \Downarrow_{\texttt{declassify}} [l : vl_L, h : vh_H]}$$

In this new SME rule, the execution is parameterized by a declassification function given by the programmer as the definition of `declassify`. In contrast to the original SME, the execution of the public view doesn't use a dummy value as the value for h. Instead h is given a value computed by `declassify` and depending on the secret value vh. In the program of Example 16, the declassification function could state that if $e.charCode$ is equal to 101 then the result of `declassify`($e.charCode$) is 101 otherwise the result is the dummy value 0. (As described in the paper [20], in more complex scenarios, a programmer annotation is required to indicate when to use the declassified value.) An implementation of the new SME mechanism is available as an extension to FlowFox [DDNP12].

Declassification in other settings: We have also studied [27] the support of relaxed noninterference by exploiting the familiar notion of type abstraction. Type abstraction in programming languages manifests in different ways; we specifically adopt the setting of object-oriented programming, where object types are interfaces, i.e. the set of methods available to the client of an object, and type abstraction is driven by subtyping. For instance, the empty interface type -the root of the subtyping hierarchy- denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data. Our key observation is that any interface in between these two extremes denotes declassification opportunities. In [**?**], we propose a method to modularly extend type systems and soundness proofs of strict information flow policies to enforce declassification. As a case study of the method, we extended the noninterference type system for the Java Virtual Machine that was developed during my PhD [17] (and extended for concurrency in [3, 8]) to enforce a declassification policy called Non-disclosure [MB09]. We conjecture that the same technique could be used to modularly extend the noninterference proof of the JavaScript hybrid type system described in Section 2.2. We have also studied declassification and access control [10] in the context of session types [HYC08], which are types for protocols. Our type system ensures secure information flow in typable protocols, including controlled forms of declassification. In particular, the type system prevents leaks that could result from an unrestricted use of session opening, selection, branching and delegation. We illustrate the use of our type system with a number of examples, which reveal an interesting interplay between our typing constraints and those used in session types to ensure properties like communication safety and session fidelity.

2.4 Confused Deputy Attacks

We have formally examined confused deputy attacks (CDAs) [Har88] in [26]. A CDA is a privilege escalation attack where a deputy (a trusted system component) can act on both its own authority and on an adversary's authority. In a CDA, the deputy is confused because it thinks that it is acting on its own authority when in reality it is acting on an attacker's authority. XSRF (see Example 12), FTP bounce attacks [CER] and clickjacking [HMW+12] are all prevalent examples of CDAs.

It is widely known that access control alone is insufficient to prevent CDAs and it is known that the use of capabilities prevents (at least some) CDAs. (In the same work, we formally show that access control and capabilities are fundamentally different: the access control semantics is strictly more permissive than the capability one.). We provide [26] the first formal definition of when a program is free from CDAs. Our definition is extensional and is inspired by information flow integrity [LMZ03], but we show that CDA-freedom is strictly weaker than information flow integrity. We formally establish that, perhaps surprisingly, capability semantics is not enough to ensure CDA-freedom. We investigate alternate approaches for CDA prevention with fewer restrictions. Similar to the work in [DMAC14], our approaches rely on provenance tracking (taint tracking).

Chapter 3

To be or not to be privacy-compliant

In which we discuss some security mechanisms for privacy and conclude that completely effective mechanisms for privacy compliance do not exist, but they should

Problem: I adhere here to the view of Saltzer and Schroeder [SS75] and other authors [AEG+17] to define privacy: *The term "privacy" denotes a socially defined ability of an individual (or organization) to determine whether, when, and to whom personal (or organizational) information is to be released.* Saltzer and Schroeder took "security" to refer to mechanisms for controlling the release or modification of information. The research in privacy can thus be divided in: (i) defining the means to allow individuals or organizations to define privacy policies and (ii) defining security mechanisms to enforce those policies. In this chapter, we are concerned with the latter problem: the definition of correct security mechanisms to enforce privacy policies.

Dream: Individuals and organizations will have the ability to determine their privacy policies and computer systems will correctly implement security mechanisms to accordingly comply with these policies.

3.1 Anonymity versus robustness

A common privacy policy is anonymity. Anonymity refers to the ability of an individual to determine that her actions should not be linked to her identity. These kind of privacy policies are formalized as anonymity properties [HM08, DLL12] and the security mechanisms that permit the compliance with these properties are known as anonymous protocols [Cha88].

Anonymity and robustness are essential properties of anonymous protocols[1]. Robustness ensures that messages reach their recipients uncorrupted and, together, anonymity and robustness ensure that protocols achieve their purported goal.

There is a large body of literature on developing protocols that achieve the desired level of privacy, and on providing quantitative measures of privacy to evaluate the relative strength of a specific protocol or to compare the strengths of two protocols [SD02, DSCP02, DLL12]. In contrast, existing definitions of robustness are scarce, and may not enforce intuitive guarantees. We have contributed to this area by providing a systematic study of the level of robustness provided by existing protocols [9]. Our goal is to define, analyze, and compare different notions of robustness for anonymity protocols, using the tools of modern cryptography. The definitions provide precise definitions of robustness that guarantee that corrupt participants will only have limited impact on the set of exchanged messages. The definitions are global, i.e. they abstract away from the internal details of the protocols; more concretely, the definitions compare the set of messages that are sent by participants and the set of messages that reach their recipients. The global character of our definitions makes them intuitive, and applicable to a variety of settings. Our robustness notions also allow us to rigorously analyze (using provable cryptography) different protocols on the same bases, as for example mixnet-based anonymous channels where the underlying mixnet does not provide strong correctness guarantees. Furthermore, our definitions fit well with *practical* anonymous channels (e.g. Tor [DMS04]) where correctness guarantees do not hold with overwhelming probability.

Example 17 (A critical review of dining cryptographers)**.** *In some cases, the existent ad-hoc correctness notions may fail to achieve the expected guarantees. To explain this important point, we consider the short dining cryptographers protocol of Golle and Juels (DC-Nets) [GJ04] for which an ad-hoc notion of robustness has been developed. We give here a high level description of the protocol, omitting technical issues that are not relevant to explain their notion of robustness.*

The protocol heavily relies on zero knowledge proofs of knowledge (ZKPoK) [GMR89]. In a nutshell, such proofs allow a prover (who possesses some piece of secret information) to convince a verifier of the veracity of a statement S, without revealing anymore information except for the validity of S. The protocol proceeds as follows.

1. *Prior to running the protocol, all n participants are allocated a different slot between* 1 *and n;*

[1]The importance of robustness in voting protocols was illustrated in our paper [9] by a fraudulous election regarding the Godfather [God]. Gilles Barthe wrote that example. As part of her work, the current author was obliged to watch that movie in order to write the paper.

2. *Each participant emits a vector of n elements that contains special values except for the one position singled out by the slot allocation protocol. In this slot the participant places his "vote" multiplied by another special value. An adversary cannot see the difference between a special value and a vote multiplied by a special value. These special $n-1$ values satisfy properties that participants must prove using a ZKPoK proof. In addition, each participant must prove that she has at least $n-1$ special values in her vector (so to prevent participants to vote twice, for example);*

3. *In order to count votes, each participant validates the vectors of all other participants by using ZKPoK tests. All vectors that do not pass the tests are considered dishonest and are discarded. For each dishonest vector, a new valid vector, with a "blank vote", is generated, for example using threshold cryptography [Ped91];*

4. *Once all vectors are validated, they are all multiplied. By properties of the special values, multiplication of all positions will "magically" cancel out, leaving a vector that contains the messages of the participants in clear.*

Without the ZKPoK proofs, the correctness guarantees of the protocol would be very low, as a corrupt participant could garble the messages of others by emitting a vector that contains trash on all positions except the one provided to him by the slot allocation protocol. Thus, the ZKPoK tests are essential to guarantee the correctness of the protocol. Indeed, Golle and Juels [GJ04] define correctness of DC-nets in terms of the (in-) ability of a corrupted participant to generate an invalid vector that will pass the ZKPoK tests. This definition of correctness is tied to the protocol phase in which the ZKPoK tests are performed, and it follows directly from the definition of ZKPoK schemes. One shortcoming of this definition of correctness is that it is intimately tied to the implementation of the cryptographic mechanisms used by a protocol, and that it does not provide end users abstract, protocol independent, guarantees.

Giving a correctness property of the protocol, a natural question then is: what does it imply? What are the robustness guarantees for the protocol? Does the correctness property entail that every message will be considered, or that every vote will be counted as expressed in the ballot? The answer is **No**. *For example, the definition of correctness of DC-Nets does not rule out a dishonest participant from emitting a valid vector that passes all ZKPoK tests and still prevents from accounting some bids or votes. One may wonder whether we can still claim that the DC-nets protocol is robust? In [9], we propose a stronger version of the protocol that is fit for being used in an election with robustness guarantees.*

Rigorous definitions While the notion of robustness had been considered elsewhere in the particular contexts of various anonymity mechanisms, the individual occurrences use ad-hoc interpretations for robustness. The correctness of an anonymity protocol can be measured by comparing the initial messages with the output messages. Consider a run of the protocol P with initial messages $\vec{M}$ and outputting the delivered messages $\vec{S}$. Note that some initial messages $\vec{M}$ may not appear in the set of delivered messages $\vec{S}$, as a result of messages being lost, intercepted, or tampered. Conversely, some of the messages delivered may not be in the set of original messages, as they may have been forged by corrupt participants or may be the result of tampering some message that was in the initial set (think for example of a corrupt participant who garbles all messages in his power). In the worst case where no appropriate measure is taken to guarantee robustness, corrupt participants may be able to force that all honest participants may be prevented to publish their messages, and all published messages originate from corrupt participants or have been tampered. This suggests two measures for robustness: the first one, which we call communication robustness, simply measures the number of messages from honest participants that are found in the result of the protocol, i.e. in the final vector of messages. The second one, which we call interference robustness, quantifies the ability of corrupt participants to prevent honest participants from publishing their messages while publishing themselves their own messages, or more generally inducing the publication of spurious messages. Informally, interference robustness intends to capture the intuition that in some protocols, like those based on DC-nets [Cha88, GJ04], corrupt participants must choose between publishing their message, or tampering (hence in some cases invalidating) a message by a honest participant. On the one hand, the communication robustness of the protocol is measured by $|\vec{M} \cap \vec{S}|$. Interference robustness of the protocol, on the other hand, is measured by $|\vec{M} \Delta \vec{S}|$ where Δ denotes the symmetric difference between two multisets.

A crucial feature of our definitions is that they are independent of the particular network topology and communication model. Indeed, our definitions of robustness are expressed only in terms of the input/output relation of the protocol, and thus are meaningful without having to specify the particular network topology and communication model. Obviously, this is not to say that the robustness of particular anonymity mechanisms can be analyzed independently of the underlying infrastructure: indeed, these details need to be spelled out and taken into account when analyzing the robustness of particular anonymity mechanisms.

Analysis of existing mechanisms We demonstrate that our definitions are very general through several examples where we quantify the robustness of existent anonymity mechanisms. The particular protocols are quite different and are intended for different scenarios. Specifically, we consider Tor [DMS04], Crowds [RR98],

a broadcast protocols using mix-nets ([PIK93, JJ01, FS01, Wik04]), and DC-nets [GJ04]. Our analyses let us conclude that for equal numbers of participants n and corrupt participants t, Mix Networks [PIK93, JJ01] guarantee the maximum communication robustness: this is n-t messages will be correctly delivered. This is the best possible communication robustness for n participants where t are corrupt. Our analyses also allowed us to quantify how the communication robustness of Tor [DMS04] depends on the probability of finding corrupted routers in the path of a message. Moreover we compared the different protocols as for the ability of a malicious sender to tamper with the message sent by an honest party in undetectable ways, i.e. interference robustness. The best possible interference robustness is 0. None of the analysed protocol satisties that. However, Mix Networks offer the stronger guarantees for interference robustness together with the stronger version of DC-Nets that we propose [9].

3.2 Privacy in web applications

Third-party services bring value to the web by enabling the trivial integration of advertisements, analytics, social networks, etc. At the same time, third-party services give rise to privacy concerns [MM12]. Indeed, third-party services can identify a user through different websites and collect this information for diverse commercial aims, *without the user's consent*. This ability to identify users is known as web tracking and is implemented via numerous technologies such as third-party cookies, HTML5 local storage, browser cache, device fingerprinting, etc. Current web tracking practices often leave users unable to determine to whom their browsing history is to be released. In order to empower users located in the European Union, a new regulation will come into action in 2018, the EU ePrivacy Regulation. This regulation will make website owners liable for third-party tracking that takes place in their websites. The penalties for non-compliance are up to 20 million euros or more[2]. In order to comply with the ePrivacy Regulation, website owners can either trust their third-party service not to track users, or exclude any third-party services, thus trading functionality for privacy. We have proposed a (partial) solution [29] for first-party servers that wish to comply with the ePrivacy Regulation *without trading functionality*. Our solution is based on the automatic rewriting of web applications in such a way that the third party requests are redirected to a trusted web server, with a different domain than the main site. A trusted server is needed so that the user's browser will treat all redirected requests as third party requests, like in the original web application. The trusted server automatically eliminates third-party tracking cookies and other technologies.

[2]Source: https://en.wikipedia.org/wiki/EPrivacy_Regulation_(European_Union)

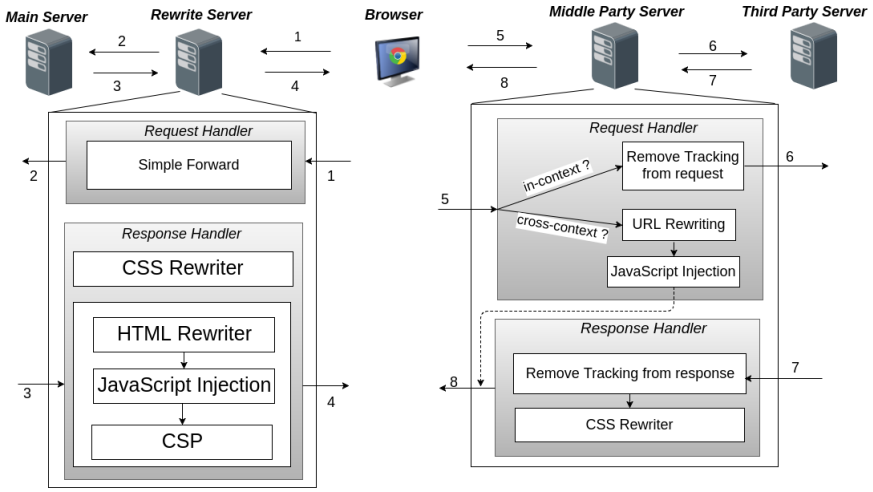


Figure 3.1: Privacy-Preserving Web Architecture

Figure 3.1 provides an overview of our web application architecture, that introduces two new components that are fully controlled by the website owner:

Rewrite Server The goal of the Rewrite Server is to rewrite the original content of the requested webpages in such a way that all third party requests will be redirected to the Middle Party Server. It consists of three main components: static HTML rewriter for HTML pages, static CSS rewriter and JavaScript injection component. Into each webpage, we inject a JavaScript code that insures that all the dynamically generated third party content is redirected to the Middle Party Server.

HTML and CSS Rewriter rewrites the URLs of static third party contents embedded in original web pages and CSS files in order to redirect them to the Middle Party Server. For example, the URL of a third-party script source `http://third.com/script.js` is written so that it is instead fetched through the Middle Party Server: `http://middle.com/?src=http://third.com/script.js`.

JavaScript Injection. The Rewrite Server also injects a script in an original webpage, that controls APIs used to inject dynamic contents. This injected script rewrites third party contents which are dynamically injected in webpages after they are rendered on the client-side.

A **Content Security Policy (CSP)** [WBV15] is injected in the response header for each webpage in order to prevent third parties from bypassing the rewriting and redirection to the Middle Party Server.

Middle Party The main goal of the Middle Party is to proxy the requests and responses between browsers and third-parties in order to remove tracking information

exchanged between them.

In-Context Contents are scripts, images, etc. Since a third party script from `http://third.com/script.js` is rewritten by the Rewrite Server to `http://middle.com/?src=http://third.com/script.js`, it is fetched through the Middle Party Server. When the middle party receives such a request URL from the browser, it takes the following steps. **Remove Tracking from request** that are set by the browser as HTTP headers. Among those headers are *Cookie, Etag, If-Modified-Since, Cache-Control, Referer*. Next, it makes a request to the third party in order to get the content of the script `http://third.com/script.js`. **Remove Tracking from response** returned by the third-party. The headers that the third party may send are *Set-Cookie, Etag, Last-Modified, Cache-Control*. **CSS Rewriter** rewrites the response if the content is a CSS file. Finally, the response is returned back to the browser.

Cross-context contents are iframes, links, popups, etc. For instance, a third party iframe from `http://third.com/page.html` is rewritten to `http://middle.com/?emb=http://third.com/page.html`. When the Middle Party Server receives such a request URL from the browser, it takes the following actions: **URL Rewriting**: instead of fetching directly the content of `http://third.com/page.html`, the Middle Party Server generates a content in which it puts the URL of the third party content as a hyperlink. `<a href = "http://third.com/page.html" rel = "noreferrer noopener"></a>`. The most important part of this content is in the `rel` attribute value. Therefore, `noreferrer noopener` instructs the browser not the send the *Referer* header when the link `http://third.com/page.html` is followed client-side. **JavaScript injection** module adds a script to the content so that the link gets automatically followed once the response is rendered by the browser. Once the link is followed, the browser fetches the third party content directly on the third party server, without going through the Middle Party server anymore. Nonetheless, it does not include the *Referer* header for identifying the website. Therefore, the `document.referrer` API also returns an empty string inside the iframe context. This prevents it from identifying the website.

This work does not address stateless tracking[3] and, as January 2018, has not been formally proven to deliver privacy guarantees. A prototype implementation can be found in [**?**].

[3] Stateless tracking refers to the identification of users by means of properties about the browser. The detection of such kind of web tracking is much more challenging than other more standard tracking based on stateful technologies such as cookies.

Chapter 4

Security of compiled programs

In which we show that, as in tango, it takes two to secure programming: a program written in a high level language must be secure as well as its low level implementation

Problem High level languages offer abstractions to the programmer. However, for security goals, the only important program is the one that executes. The problem is that program abstractions may hold security guarantees at a high level but implementations may not preserve the same guarantees after compilation.

The Dream For there to be any hope of building secure and correct programs, developers need to understand on what program's behaviour they can rely on. In an ideal world programs would be written using languages with a mathematically precise semantics. These languages would provide simple abstractions to specify behaviour and security requirements and those requirements would be met by compiler implementations.

4.1 Breaking abstractions

High level programming languages allow the programmer to reason about programs using simple abstractions. After the program is written, it is compiled; but ... *Can you trust your compiler?* This is the first sentence of the paper describing the major effort -up-to-date- in the area of compiler correctness, CompCert, a C compiler fully verified for correctness [Ler06]. Compiler correctness stipulates that the compiled code should behave as prescribed by the semantics of the source program, thus preserving every trace property of the source code. Hence, you can trust CompCert to preserve the behaviour of source programs. Yet, we can refine the question and ask about a correct compiler: *Can you trust your compiler to preserve security properties?* Even if the compiler is correct, this question might not have the same

answer as the first one. The reason is that compiler correctness guarantees that trace properties are preserved, but often, security properties are hyperproperties [CS10, CC08] and include more than a trace at the time, shaping (partial) equivalences between programs traces. Thus, in order to positively answer the second question, not only we need a correct compiler but also a secure one, that is, a compiler that preserves security properties. Although the area of secure compilation is not new [Aba98, AFG98], several fairly recent efforts exist towards the study and the development of secure compilers (see [CSH09, DPPK17, SP16b, JHdA+16] to cite just a few). In the following subsections, I describe a few efforts which belong to my research.

Unfortunately, a correct and secure compiler is still not enough to guarantee the security of compiled code. This has been violently attested by striking recent news on the Spectre and Meltdown attacks [Spe]: breaking programming abstractions can also be the fault of hardware inconsistencies with the stipulated low level semantics.

4.2 Secure distributed abstractions

In a distributed system values are shared and computed by programs at different locations. This requires attention to many implementation details such as the preservation of invariants of shared values, the implementation of the desired control flow between distributed programs, the preservation of control-flow integrity, and for robustness against malicious networks and active adversaries, cryptographic protocols to protect information and cryptographic key exchanges. Each of the mentioned implementation details requires the programmer expert knowledge without which programming distributed systems can be prone to incorrect and insecure implementations.

In particular, when considering the integrity and confidentiality of information, the verification of distributed programs entangles different aspects of system implementations, ranging from application-level information-flow control down to cryptographic algorithms and communication protocols. We have studied secure compilers [5, 7, 13] that implement security abstractions based on information flow security. Technically, this involves the integration between language-based security and protocol verification techniques.

As partly described in Chapter 2, in information flow security, confidentiality and integrity policies are specified using security labels, equipped with a partial order that describes permitted flows of information.Security labels associated to program variables specify who can read from (confidentiality) and who can write to (integrity) a given variable. The preservation of confidentiality and integrity policies is expressed as *noninterference* properties, guaranteeing that the knowledge of an adversary with limited access to variables is not augmented by any program

execution.

We consider cryptographic enforcement mechanisms for confidentiality and integrity in distributed programs. Our security model accounts for active adversaries, represented as untrusted (or unknown) parts of the program that may change unprotected memory during execution. The resulting programs may represent, for instance, distributed systems connected by some untrusted network, or untrusted machines containing protected subsystems. According to the program semantics, security depends on an abstract read/write policy for accessing shared memory. In their cryptographic implementation, shared memory is unprotected, and security depends instead on encryption and signing when accessing the shared memory.

Example 18 (A basic example)**.** *Consider two parties a and b that wish to perform some computation securely by exchanging a series of messages over some untrusted network. Using shared memory and imperative programs as distributed abstractions, we may write for instance*

$$\_\,;(x := 1)^a;\_\,;(\mathsf{if}\ x = 1\ \mathsf{then}\ y := 2\ \mathsf{else}\ y := z)^b;\_\,;(y := y + 1)^a;\_$$

where the parentheses $(\,)^a$ and $(\,)^b$ indicate pieces of code that runs on behalf of a and b, respectively, and where the placeholders $\_$ stand for any untrusted code that may run in-between. (This unknown, untrusted code represents some active adversary.) In this example, we have three trusted pieces of code that share variables x, y, and z and, if we assume that untrusted code cannot access these variables, we would expect, for instance, that z remains secret and $y = 3$ at the end of any run. To meet our expectations, a needs to securely pass x to b, then b needs to securely pass y to a.

In a less abstract setting, such flows of information between shared variables may involve communication over untrusted channels. We model these communications using untrusted shared memory, that is, memory that can be read and written by active adversaries, with some adequate encryption and signing. For example, we may use variables x_e, x_s and y_e, y_s to pass the encrypted values and signatures for x and y, respectively, and the second command

$$(\mathsf{if}\ x = 1\ \mathsf{then}\ y := 2\ \mathsf{else}\ y := z)^b$$

may be implemented as

$$\begin{array}{l}\mathsf{if}\ \mathcal{V}(x_e, x_s, k_v)\ \mathsf{then}\ (\\ \quad x'_e := x_e; x^b := \mathcal{D}(x'_e, k_d);\\ \quad \mathsf{if}\ x^b = 1\ \mathsf{then}\ y^b := 2\ \mathsf{else}\ y^b := z^b;\\ \quad y'_e := \mathcal{E}(y^b, k_e); y_s := \mathcal{S}(y_e, k_s); y_e := y'_e)\end{array}$$

where, in order to read x out of its wire format x_e, x_s, the code first verifies ($\mathcal{V}$) the signature then performs a decryption ($\mathcal{D}$) to extract a local copy of x into x^b; and, conversely, for writing y, the code first encrypts ($\mathcal{E}$) its updated local copy y^b and then signs ($\mathcal{S}$) the encrypted value.

This implementation code does not rely on the confidentiality or integrity of the shared variables used on the wire (variables x_e, x_s, y_e, y_s). Instead, it relies on local variables (x'_e, x^b, y^b, z^b, y'_e), on the adequate generation and management of the keys used for verifying, decrypting, encrypting, and signing (variables k_v, k_d, k_e, and k_s), and on security assumptions on their respective cryptographic primitives.

If we use a public-key signature scheme $(\mathcal{S}, \mathcal{V})$, for instance, the integrity of the verification key k_v must be higher than the integrity of x, while the confidentiality of the signing key k_s must be high enough to protect the integrity of y. Also, if the keys are used for other purposes (and we can hardly dedicate 4 keys to every variable assignment), we need to carefully control their interaction. For instance, in the code above, we cannot use the same keys for protecting both x and y, since an adversary may then achieve $y = 2$ at the end of the computation by inserting the code $y_e := x_e; y_s := x_s$ between b and a. Besides, we cannot assume that the computation always completes successfully, as indeed an adversary may insert $x_s := 0$ before b's code and thus cause the signature verification at the beginning of our code above to fail, so we also need to qualify our notion of integrity for this computation.

We enforce information flow policies in programs that run at multiple locations, with diverse levels of security. This involves cryptographic protection whenever relatively secure locations (e.g. a client and a server) interact via less secure locations (e.g. an open network).

To this end, we compile programs down to distributed implementations that run under the control of the adversary.

- In source programs, security depends on a global program semantics, with abstract policies for reading and writing shared memory. These policies enable a simple review of confidentiality and integrity properties.
- In their distributed implementations, shared memory is unprotected, the adversary controls the scheduling, and security depends instead on cryptographic protection.

Our secure compiler [Tool:Cflow, 5, 7] is structured into four stages: slicing, control flow, replication, and cryptography. The first stage slices sequential code with locality annotations into a series of local programs, each meant to run at a single location. After slicing, the second stage protects the control flow of the source program against a malicious scheduler, by generating code that maintains auxiliary variables to keep track of the program state, based on its integrity policy. The replication

stage transforms a distributed program (still relying on a global, shared, protected memory) into a program where variables are implemented as local replicas at each location, with explicit updates between replicas. Finally, the cryptography stage inserts cryptographic operations to protect these variable updates, and it generates an initial protocol for distributing their keys.

Our target notions of security are expressed in a computational model of cryptography. In this model, adversaries are probabilistic programs that operate on bitstrings and have limited computational power (so that they cannot effectively break cryptography by brute force). This leads us to reason with polynomial-time hypotheses and probabilistic semantics. We could have used instead a symbolic model of cryptography, where adversaries may perform arbitrary computations on abstract algebraic terms (not bitstrings). However, this simpler model would have hidden many cryptographic side channels (see also Section 1.1) that are relevant in distributed implementations and problematic for information security. The relation between symbolic and computational models is the subject of active research [**?**, see e.g.]]aba:rog:02,bac:pfi:wai:03, soundequivs but it is unlikely that they can be reconciled at the level of details handled by our compiler. Thus, we seek computational soundness directly for information-flow security, rather than for symbolic cryptography.

4.3 Mashup security by compilation

In order to quickly integrate third-party code to augment functionality in mashups, programmers often use the script tag, granting unnecessarily privileges to untrusted code (see Sect.2.1.2 in Chapter 2). What tools could help programmers to still quickly integrate third-party code while complying to the principle of least privileges [BSS11]?

In 2009, Barth et.al [BJL09] proposed Postmash, a designed mashup architecture to use inter-frame communication between integrator and gadgets. PostMash architecture includes stub libraries on both the integrator and the gadget. On the integrator side, the stub library must provide an interface similar to the original gadget's interface (See e.g. Example 10, the `gadget` object). The stubbed interface sends corresponding messages by means of the PostMessage API in HTML5. On the gadget side, there is another stub library, listening and decoding incoming messages. The Postmash design lacks generality since each gadget requires different interfaces and it requires programmer intervention: the programmer manually writes interfaces for different gadgets. Moreover, the programmer needs to write integrator's code in CPS (Continuation Passing Style)[DF92] to adapt to the asynchronous nature of PostMessage. These issues compromise the guarantees offered by the architecture. Crucially, reliability of the mashup depends on the programmer

without any enforced guarantees. We have investigated [15] the following questions about the PostMash design:

1. Can the stub libraries be made general (the same libraries for every gadget and integrator)?

2. Can PostMash mashups be automatically generated starting from potentially insecure mashups and preserving only the good behaviour of the original mashup?

3. Is it possible to precisely define the security guarantees offered by the architecture?

We have positively answered these questions. We address questions 1 and 2 with a compiler called Mashic [Tool:Mashic] which inputs existing mashup code, JavaScript code integrated to HTML, to generate reliable mashups using gadget isolation.For question 2, we formalize the notion of "benign gadget" that is useful to prove precisely in which cases the generated mashup behaves as the original one. The answer to question 3 corresponds to a formalization (in the shape of an observational semantics equivalence) of the security guarantees offered by the Same Origin Policy in a browser. The Mashic compiler [Tool:Mashic] offers the following features:

Automation and generality: Inter-frame communication and sandboxing code is fully generated by the compiler and can be used with any untrusted gadget without rewriting the gadget's code. After sandboxing, gadget objects are not directly reached by the integrator when the SOP applies. Instead the integrator uses opaque handles [Vin97] to interact with the gadget. Due to the asynchronous nature of the PostMessage API, integrator's code is transformed into CPS.

Correctness guarantees: We prove a correctness theorem that states that the behavior of the Mashic compiled code is equivalent to the original mashup behavior under the hypothesis that the gadget is *benign* and a correctness hypothesis of marshaling/unmarshaling for objects that are sent via postMessage. Precisely defining a benign gadget turned out to be a technical challenge in itself. For that, we instrument the JavaScript semantics extended with HTML constructs by a generalization of colored brackets [GMZ00] and resort to equivalences used in information flow security [SM03].

Security guarantees: We prove a security theorem that guarantees a delimited form of integrity and confidentiality for the compiled mashup. Information sent from the integrator to the gadget, corresponds to a declassification. We prove that the gadget cannot learn more than what the integrator sends. Analogously, the influence that the gadget can have on the integrator is delimited to the actions that the integrator performs with the messages that the gadget sends to the integrator.

Example 19 (Mashic compilation)**.** *Recall the mashup presented in Example 11. After compilation with Mashic, instead of directly including the script, a new origin* `u-i.com` *is used as an untrusted gadget container, and the gadget code is put in an iframe belonging to this origin.*

By doing this, the JavaScript execution environment between integrator and gadget is isolated, as guaranteed by the browser's SOP. Limited communication between frames and integrator is possible through the PostMessage API in the browser if there is an event listener for the 'message' event. To register a listener one provides a callback function as parameter and treats messages in a waiting queue, asynchronously. With PostMessage, only strings can be sent. However, it is possible to marshal objects that do not point to themselves (as e.g. the global object), via a marshaling method, such as the standard JSON stringify.

Code in `adservice.js` *and* `trusted.js` *needs to adapt to the asynchronous behaviour. Instead of calling methods or functions, the integrator must send messages to manipulate the untrusted gadget as shown in the following example:*

```
PostMessage(stringify({action : "newInstance",
              container : "gadget_div",
              type : "SIMPLE"}),
             "http://u-i.com");
PostMessage(stringify({action : "setLevel",
              container : "gadget_div"}),
              "http://u-i.com");
```

Compilation with Mashic will not preserve the malicious behavior of Listing **??** *but will only preserve behavior that does not represent a confidentiality or integrity violation to the integrator.*

The proposed compiler is directly applicable to real world and widespread mashups. We present evidence that our compiler is effective. We have compiled several mashups based on Google and Bing maps, YouTube, and Zwibbler APIs. In order to reduce the overhead of using Mashic-compiled code, we have proposed an optimization [23] for reducing the cost of cross-domain operations between gadgets and integrator in which an automatic batching mechanism groups together cross-domain operations in straight-line code, supporting loops and branching instructions.

4.4 Secure program abstractions for the web

Programming a web application as a single code for the server and the client, and written in a single unified language, is known as multitier programming. Multitier languages have been proposed in response to the emergent need of simplifying the development process of web applications. Examples of such languages include

Hop [SGL06, SP16a], Links [CLWY06, BG09], Swift [CLM+09], SELinks [CSH09], and Ur [Chl10].

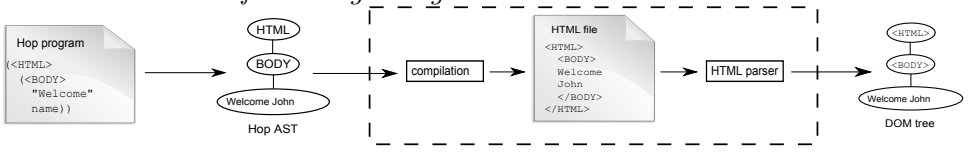
We formally provide a formal and unified small-step operational semantics for reasoning about web applications written in Hop [11, 16]. On one hand, the Hop language relies on standard programming constructs. On the other hand, several features of Hop are specific to a multitier language, and therefore require specific semantics. In particular, the dynamic client code generation from the server and its installation at client site (see Section 2.1.1).

Multitier languages provide natural tools to solve code injection problems, as they allow *global* reasoning for web applications. We propose a methodology for preventing code injection in multitier languages [12], which consists in modifying the client code compiler at the point of dynamic generation for comparing the generated code with the specification extracted from the syntax of the source program.

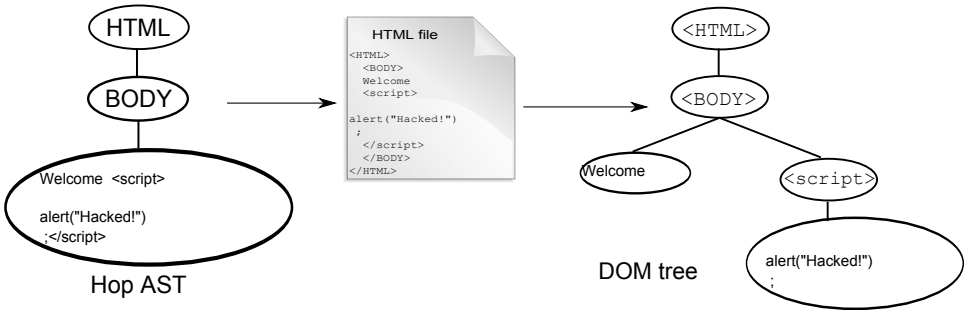
Example 20 (Code injection detection in HOP)**.** *Recall the program in Example 9 in Sect.2.1.1. In a multitier language, an equivalent program can be written using constructs from the language, as shown in the following Hop example:*

```
(define-service (name)
(<HTML>
   (<BODY>
       "Welcome" name )))
```

As in Example 9, the code will respond to a client request by a HTML page containing a greeting, but the difference is that `<HTML>` *and* `<BODY>` *are regular library functions of the language. Therefore the Hop Abstract Syntax Tree (AST) obtained from the server-side run-time environment should have the same structure from the DOM tree obtained by parsing the generated HTML document in the client's browser, as shown in the following image where* `name` *has been instantiated with "John" :*



In the case of a code injection attack, the input `name` *binds to a malicious string* `"<script>...</script>"`*. Therefore the Hop AST obtained from the server-side run-time environment is different from the DOM tree obtained by parsing the generated HTML document in the client's browser.*



Being provided with a server-side Hop AST makes code injection detection easier: it is only needed to reproduce the DOM tree that is generated on the client-side and compare it with the Hop AST. This tree can be easily obtained by parsing the HTML document generated from the Hop AST on the server-side *using a standard HTML parser. Then it is sufficient to compare the two trees to detect code injection attacks. If the two trees have the same structure, the program is safe and the response is sent to the client. If the two trees differ, code has been injected and an exception is raised.*

The methodology follows the technique for SQL injection by Su and Wasserman [SW06]. The methodology complements that of [SW06] by the following added value:

- The programmer is freed from making any intervention in order to achieve security guarantees. Indeed the expected syntax structure is not provided by the programmer, since it is already given by the syntax of the multitier program.

- Proofs are given by means of standard language-based techniques and programming language semantics. In order to prove that our methodology eliminates unexpected behaviors from the dynamically generated pages, we use the Hop program semantics [11, 16], which abstracts away from compilation processes, as the specification of what expected behavior is. We formally prove the validity of our approach by showing that the client compiler is fully abstract.

Interestingly, multitier language semantics and compilers provide a natural formal **definition of code injection freedom**: a program is free from code injection if the source semantics includes all the behaviours of compiled code (Notice that this represents the inverse simulation stipulated by program correctness). When the compiler is the client-code compiler of a multitier language, the definition corresponds to XSS freedom. The tree comparison technique used in Hop [12] can, in principle, also be applied to other traditional programming languages for Web applications, with additional efforts. For example, to apply this technique on a Web server that supports PHP requires to modify the Web server in a non-trivial way:

1. For each PHP program that generates a HTML page, associate it with a separate function that computes an AST as specification depending on given inputs.

2. Upon receiving a HTTP request from a client of a PHP program, the Web server invokes the corresponding specification function on received input, obtaining an AST. It then executes the PHP program with a PHP interpreter, and parses the output of the program with a HTML parser, obtaining another HTML tree. The server delivers the HTML output only if the trees are of the same shape.

Comparing to the multitier programming languages approach, applying tree comparison technique on traditional programming languages for the Web requires further modification on Web servers and programmers' intervention to write a specification of their code.

However, the tree comparison technique alone is **insufficient** to prevent all kind of code injection attacks in practice, as noted in [RL12]. Indeed, a code is considered to be injected when a user input string ends up being interpreted by the browsers as a client-side expression. To prevent this Hop imposes that all client-side code, coming from the first party server, should be generated by the Hop client-side compiler.

This restriction is difficult to impose in a non-multitier language, such as PHP because in HTML we can include JavaScript expressions in the HTML nodes attributes. Hop handles this situation by a simple filtering that rejects all attributes of nodes. In the HTML specification, those attribute strings as event handler will be interpreted as script expressions. Hop imposes that these attributes are bound to Hop client-side expressions, not to strings. For instance, it rejects

```
(<DIV> :onclick "alert( msg )" ...)
```

but it accepts

```
(<DIV> :onclick ~(alert msg) ...)
```

where (), called tilde code expression, is a Hop function to represent client code. Since Hop offers no means for transforming a string of characters into a *tilde code* expression, this simple filtering technique ensures that attributes as event handlers cannot be used to inject arbitrary expressions.

Beyond the HTML specification, most browsers interpret attributes values prefixed with the string `javascript:` as listener attributes. For instance, the following HTML link, when clicked, evaluates the `alert` function call.

```
<A href="javascript:alert('foo')">click me</a>
```

This could also lead to code injection. To solve that problem we have adopted a conservative solution that disables this extension by forbidding the `javascript:` prefix for all attributes. This is enforced by Hop for HTML trees as well as for CSS declarations.

Finally, Hop also ensures that pure client-side manipulation cannot yield to executing new code. For that, the client-side runtime library only binds JavaScript functions that are safe. For instance, if functions such as `eval` or `document.write` were accessible in Hop they could be used to evaluate arbitrary user forged code. The dangerous functions are either not included or slightly modified. For instance facilities such as `innerHTML` that parses an inserted string to a HTML tree is kept, but its argument must be a HTML node instead of a string.

One More Thing ....

There is one last thing[1], or actually many things, to mention as perspectives. A new industrial revolution is here[2], the Internet of Things (IoT). The Internet has expanded to connect not only computers but all kind of physical objects (a.k.a."smart" things). The IoT is changing the way humans interact with physical objects. What is the role of researchers in security in this revolution? As with the Internet, IoT connectivity brings associated security problems and poses a danger to people's privacy, but in contrast to problems before this new revolution, IoT security issues have the power to easily endanger people's life. In 2015, for example, researchers[JEE] discovered that the Jeep Cherokee could be hacked over the Internet, causing unintended acceleration and slamming on the car's brakes or turning the vehicle's steering wheel at any speed. Concerns about IoT security put this whole revolution at stake[3].

IoT specificities create unique challenges to be addressed in order to tackle the problem of IoT security in its entirety. IoT specificities include:

- **constrained resources:** microcontrollers in IoT devices are architecturally very different in terms of storage and CPU capabilities compared to a typical PC. Certain capabilities like RAM, Flash, word size, and CPU Speed are several orders of magnitude lower than in classical architectures [**?**]. The constraints make the use of certain classical security building blocks, such as standard cryptographic libraries and protocols [Bog17, IOT], not adapted for microcontrollers.

- **unprecedented heterogeneous nature:** covering all the spectrum from microcontrollers to the cloud, an IoT application potentially include code that runs in web clients, servers, and various different IoT devices with different architectures. Moreover, IoT applications must handle a wide variety of asynchronous events, launching calculations that trigger a cascade of new events.

[1] In honor to Steve Jobs that I admire for his power to innovate and also for his power to make other people innovate.

[2] Source: the World Economic Forum

[3] The dream is to survive to it.

As happened with programming of mobile applications, programming of the IoT is not reserved for expert developers but it is opened for everyone. For example, Samsung SmartApps is a smart home programming framework to program your door locks, oven, and other home appliances. Recently, researchers [FJP16] have discovered that applications in this framework can gain access to more operations on devices than their functionality requires.

There is a need to simplify the development process of IoT applications and provide IoT programming abstractions and analyses to handle the challenges of IoT secure programming.

List of my publications 2007-2017

[1] Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security Types Preserving Compilation. *Journal of Computer Languages, Systems & Structures*, 33(2), 2007.

[2] Gilles Barthe, David Pichardie, and Tamara Rezk. A Certified Lightweight Noninterference Java Bytecode Verifier. In *Programming Languages and Systems, 16th European Symposium on Programming (ESOP)*, 2007.

[3] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of Multithreaded Programs by Compilation. In *12th European Symposium on Research in Computer Security (ESORICS)*, 2007.

[4] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable Enforcement of Declassification Policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008.

[5] Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information Flow Security. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[6] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate Translation for Optimizing Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5), 2009.

[7] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A Security-Preserving Compiler for Distributed Programs. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[8] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of Multithreaded Programs by Compilation. In *ESORICS'07 Special Issue in ACM Transactions on Information and System Security (TISSEC)*, 2010.

[9] Gilles Barthe, Alejandro Hevia, Zhengqin Luo, Tamara Rezk, and Bogdan Warinschi. Robustness Guarantees for Anonymity. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, 2010.

[10] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session Types for Access and Information Flow Control. In *Concurrency Theory, 21th International Conference (CONCUR)*, 2010.

[11] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Towards Reasoning about Web Applications: An Operational Semantics for HOP. *APLWACA*, 2010.

[12] Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Automated Code Injection Prevention for Web Applications. In *Theory of Security and Applications (TOSCA)*, 2011.

[13] Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information Flow Types for Homomorphic Encryptions. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[14] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure Information Flow by Self Composition. *Mathematical Structures in Computer Science (MSCS)*, 21(6), 2011.

[15] Zhengqin Luo and Tamara Rezk. Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication. In *25th IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[16] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Reasoning about Web Applications: An Operational Semantics for HOP. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2), 2012.

[17] Gilles Barthe, David Pichardie, and Tamara Rezk. A Certified Lightweight Noninterference Java Bytecode Verifier. *Mathematical Structures in Computer Science (MSCS)*, 23(5), 2013.

[18] Ana Gualdina Almeida Matos, José Fragoso Santos, and Tamara Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *Trustworthy Global Computing - 9th International Symposium (TGC)*, 2014.

[19] José Fragoso Santos and Tamara Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference (SEC)*, 2014.

[20] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful Declassification Policies for Event-Driven Programs. In *IEEE 27th Computer Security Foundations Symposium (CSF)*, 2014.

[21] José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt. Hybrid Typing of Secure Information Flow in a JavaScript-like Language. In *Trustworthy Global Computing - 10th International Symposium (TGC)*, 2015.

[22] José Fragoso Santos, Tamara Rezk, and Ana Almeida Matos. Modular Monitor Extensions for Information Flow Security in JavaScript. In *Trustworthy Global Computing - 10th International Symposium (TGC)*, 2015.

[23] Zhengqin Luo, José Fragoso Santos, Ana Almeida Matos, and Tamara Rezk. Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication. *Journal of Computer Security*, 24(1), 2016.

[24] Nataliia Bielova and Tamara Rezk. A Taxonomy of Information Flow Monitors. In *Principles of Security and Trust - 5th International Conference (POST)*, 2016.

[25] Nataliia Bielova and Tamara Rezk. Spot the Difference: Secure Multi-execution and Multiple Facets. In *21st European Symposium on Research in Computer Security (ESORICS)*, 2016.

[26] Vineet Rajani, Deepak Garg, and Tamara Rezk. On Access Control, Capabilities, their Equivalence, and Confused Deputy Attacks. In *IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016.

[27] Raimil Cruz, Tamara Rezk, Bernard P. Serpette, and Éric Tanter. Type Abstraction for Relaxed Noninterference. In *31st European Conference on Object-Oriented Programming (ECOOP)*, 2017.

[28] Raimil Cruz, Tamara Rezk, Bernard P. Serpette, and Éric Tanter. Type Abstraction for Relaxed Noninterference (artifact). *DARTS*, 2017.

[29] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. Control What You Include! - Server-Side Protection Against Third Party Web Tracking. In *Engineering Secure Software and Systems - 9th International Symposium (ESSoS)*, 2017.

[30] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the Content Security Policy Violations due to the Same-Origin Policy. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*, 2017.

Software related to my research 2007-2017

[Tool:Cflow] The CFlow secure compiler. http://www.le-guernic.info/cflow.html.

[Tool:Mashic] The Mashic secure compiler. http://web.ist.utl.pt/~ana.matos/Mashic/mashic.html.

[Tool:Iflowsig] The Iflowsig inlining information flow control compiler. https://www.doc.ic.ac.uk/~jfaustin/IFMonitor/index.html.

[Tool:Iflowtypes] The Iflowtypes JavaScript type checker. https://www.doc.ic.ac.uk/~jfaustin/TGCTypeSystem/index.html.

[Tool:RNIchecker] Type abstraction for relaxed noninterference. https://pleiad.cl/research/software/obsec.

[Tool:Webstats] Webstats. http://webstats.inria.fr.

[Tool:WTArch] Web tracking protection architecture. http://www-sop.inria.fr/members/Doliere.Some/essos/deployment.html.

> Beware of bugs in the above code; I have only proved it correct [*and, in our case, secure*], not tried it.
>
> *Donald Knuth, 1977*

Bibliography

[Aba98] Martín Abadi. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming*, 1998.

[ABHS09] Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness and completeness of formal encryption: The cases of key cycles and partial information leakage. *Journal of Computer Security*, 2009.

[Ada78] Douglas Adam. *The Hitchhiker's Guide to the Galaxy*. 1978.

[AEG+17] Martín Abadi, Úlfar Erlingsson, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Nicolas Papernot, Kunal Talwar, and Li Zhang. On the protection of private information in machine learning systems: Two recent approches. In *30th IEEE Computer Security Foundations Symposium, CSF*, 2017.

[AF10] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, 2010.

[AF12] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th Symposium of Principles of Programming Languages*. ACM, 2012.

[AFG98] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.

[AR02] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 2002.

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth

Smith. A trusted mechanised javasript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2014.

[BG09] Ioannis G. Baltopoulos and Andy Gordon. Secure compilation of a multi-tier web language. In *Workshop on Types in language design and implementation, TLDI*, 2009.

[BJL09] Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *W2SP2009*, 2009.

[Bog17] Andrey Bogdanov, editor. *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*, volume 10098 of *Lecture Notes in Computer Science*. Springer, 2017.

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. *IACR Cryptology ePrint Archive*, 2002.

[BS14] Gérard Berry and Manuel Serrano. Hop and hiphop: Multitier web orchestration. In *Distributed Computing and Internet Technology - 10th International Conference, ICDCIT*, 2014.

[BSS11] David A. Basin, Patrick Schaller, and Michael Schläpfer. *Applied Information Security - a Hands-on Approach*. Springer, 2011.

[CC08] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *ACM Conference on Computer and Communications Security, CCS*, 2008.

[CER] CERT. Ftp bounce attacks.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1988.

[Chl10] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *International Conference on Programming Languages and Implementation (PLDI)*, Toronto, Canada, June 2010. ACM Press.

[CLM+09] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Communications of the ACM*, 52(2):79–87, 2009.

[CLWY06] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, November 2006. Springer.

[CRB17] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. CCSP: controlled relaxation of content security policies by runtime policy composition. In *26th USENIX Security Symposium, USENIX Security*, 2017.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.

[CSH09] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *ACM SIGMOD International Conference on Management of Data, SIGMOD*, 2009.

[DDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a Web Browser with Flexible and Precise Information Flow Control. In *CCS*, 2012.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *MSCS*, 2(4), 1992.

[DLL12] Jannik Dreier, Pascal Lafourcade, and Yassine Lakhnech. A formal taxonomy of privacy in voting protocols. In *IEEE International Conference on Communications, ICC*, 2012.

[DMAC14] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *IEEE 27th Computer Security Foundations Symposium, CSF*, 2014.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[DP10] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, 2010.

[DPPK17] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. Modular, fully-abstract compilation by approximate back-translation. *Logical Methods in Computer Science*, 13(4), 2017.

[DSCP02] C. Díaz, S. Seys, J. Claessens, and B. Preneel. Towards measuring anonymity. In *Privacy Enhancing Technologies Workshop, PET*, 2002.

[ECM] ECMAscript 2017 language specification. https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. Accessed: 2018-01-28.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, 1984.

[FJP16] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy, SP*, 2016.

[Fla06] Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2006.

[FOPS01] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference*, 2001.

[FS01] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer, 2001.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM symposium on Theory of computing (STOC)*, 2009.

[GJ04] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *EUROCRYPT*, 2004.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, 1982.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 1988.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[GMZ00] Dan Grossman, J. Gregory Morrisett, and Steve Zdancewic. Syntactic type abstraction. *TOPLAS*, 22, 2000.

[God] The godfather. https://en.wikipedia.org/wiki/The_Godfather. Accessed: 2018-01-28.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference*, 2010.

[Har88] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 1988.

[Har17] Yuval Noah Harari. *Homo Deus*. 2017.

[HM08] Alejandro Hevia and Daniele Micciancio. An indistinguishability-based characterization of anonymous channels. In *Privacy Enhancing Technologies Symposium, PETS*, Lecture Notes in Computer Science, 2008.

[HMW+12] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, 2012.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2008.

[IOT] Iot seamless personal authentication. http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/onlinesecurityprize-01-2017.html.

[JEE] Jeep cherokee attack. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[JHdA+16] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF*, 2016.

[JJ01] Markus Jakobsson and Ari Juels. An optimally robust hybrid mix network. In *ACM symposium on Principles of distributed computing, PODC*, 2001.

[JJLS10] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS*, 2010.

[Joh14] Martin Johns. Script-templates for the content security policy. *J. Inf. Sec. Appl.*, 19, 2014.

[KHFS09] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. Directing javascript with arrows. In *Proceedings of the 5th Symposium on Dynamic Languages, DLS*, 2009.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages, POPL*, 2006.

[LGBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based Confidentiality Monitoring. In *Proc. of the Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

[Lip81] Richard J. Lipton. How to cheat at mental poker. In *Proceedings of the AMS short course on Cryptology*, 1981.

[LMZ03] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *1st Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.

[LZ05] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Symposium on Principles of Programming Languages (POPL)*, 2005.

[Mar04] Bruno Martin. *Codage, cryptologie et applications*. Presses polytechniques et universitaires romandes, 2004.

[MB09] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 2009.

[MF09] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 2009.

[MM12] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy, SP*, 2012.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008*, 2008.

[MMT09] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Computer Security - ESORICS*, 2009.

[MT09] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF*, 2009.

[Nar14] Francesco Zappa Nardelli. *Reasoning between programming languages and Architectures (HDR)*. 2014. http://www.di.ens.fr/~zappa/readings/hdr/.

[NFR+18] Mihn Ngo, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. A better facet of dynamic information flow control. In *The Web Conference (WWW)*, 2018. to appear.

[NPR18] Mihn Ngo, Frank Piessens, and Tamara Rezk. Impossibility of precise and sound termination sensitive security enforcements. In *IEEE Symposium on Security and Privacy*, 2018. to appear.

[OSI05] Rafail Ostrovsky and William E. Skeith III. Private searching on streaming data. In Victor Shoup, editor, *Advances in Cryptology–CRYPTO 2005*, LNCS, 2005.

[OWA] OWASP. https://www.owasp.org.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[Ped91] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91*, Lecture Notes in Computer Science, 1991.

[PIK93] C. Park, K. Itoh, and K. Kurosawa. Efficient anonymous channel and all/nothing election scheme. In Tor Helleseth, editor, *Advances in Cryptology—EUROCRYPT 93*, pages 248–259, 1993.

[RL12] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2012.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, November 1998.

[RS91] Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO*, 1991.

[SCB16] Alvise Rabitti Stefano Calzavara and Michele Bugliesi. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In *ACM Conference on Computer and Communications Security, CCS*, 2016.

[SD02] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *Privacy Enhancing Technologies Workshop, PET*, 2002.

[SGL06] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the web 2.0. In *Dynamic Languages Symposium (DLS)*, 2006.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[SOP] Same origin policy. https://www.w3.org/Security/wiki/Same_Origin_Policy.

[SP16a] Manuel Serrano and Vincent Prunet. A glimpse of hopjs. In *International Conference on Functional Programming, ICFP*, 2016.

[SP16b] Raoul Strackx and Frank Piessens. Developing secure sgx enclaves: New challenges on the horizon. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, 2016.

[Spe] Spectre and Meltdown attacks. https://spectreattack.com/. Accessed: 2018-01-30.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[SS09] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2009.

[SSM10] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web, WWW*, 2010.

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL*, pages 372–382, 2006.

[Thi05] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP , ETAPS*, 2005.

[Vin97] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[WBV15] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Candidate Recommendation, 2015.

[Wik04] Douglas Wikström. A universally composable mix-net. In *TCC*, 2004.

[WSLJ16] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *ACM Conference on Computer and Communications Security, CCS*, 2016.

[Zda02] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.